

Runtime Polymorphic Generic Programming—Mixing Objects and Concepts in ConceptC++

Mat Marcus¹ Jaakko Järvi² Sean Parent¹

¹Adobe Systems Inc. (`{mmarcus|sparent}@adobe.com`)

²Texas A&M University (`jarvi@cs.tamu.edu`)

2007-07-31

Setting

- ▶ Generic programming (in the style of STL) is a proven paradigm for developing reusable libraries
 - ▶ Non-intrusive
 - ▶ Static polymorphism (only)
- ▶ Object-oriented programming
 - ▶ Intrusive
 - ▶ Runtime polymorphism (only)
- ▶ This work: combination of both leads to non-intrusive runtime polymorphism (good thing)
 - ▶ Presented techniques achieve modularity of components in Adobe source libraries: successfully integrated into several Adobe applications
- ▶ Novel idioms for library design and implementation in ConceptC++

Outline

- ▶ Static polymorphism in generic programming
- ▶ Dynamic polymorphism in object-oriented programming
- ▶ Basics of ConceptC++
- ▶ Approach to combine best of both worlds by instantiating generic components with non-intrusive run-time polymorphic types
- ▶ Implementation details of the approach (very little)
- ▶ Conclusions

Static polymorphism of generic programming

```
template <typename P>
struct layout_engine {
    void append(P placeable);
    void solve() {
        ... measure(placeables_m[i], extents_m[i]);
        // solve layout constraints and update place_data_m
        ... place(placeables_m[i], place_data_m[i]); ...
    }

    vector<extents_t> extents_m;
    vector<P> placeables_m;
    vector<place_data_t> place_data_m;
}
```

- ▶ `measure` and `place` must be overloaded for `P` (with the right signature and semantics)

Using the layout engine with a specific widget type

```
template <typename P> struct layout_engine { ... };

void measure(HIViewRef& t, extents_t& result) { ... };
void place(HIViewRef& t,
           const place_data_t& place_data) { ... };

layout_engine<HIViewRef> le;
HIViewRef w;
...
le.append(w);
...
le.solve();
```

Polymorphism via indirection

```
struct layout_engine {
    void append(PlaceableBase* placeable);
    void solve() {
        ... placeables_m[i]->measure(extents_m[i]);
        // solve layout constraints and update place_data_m
        ... placeables_m[i]->place(place_data_m[i]); ...
    }

    vector<extents_t> extents_m;
    vector<PlaceableBase*> placeables_m;
    vector<place_data_t> place_data_m;
}
```

Using the layout engine with many widget types — Reference semantics

```
struct MyWidget : PlaceableBase {  
    virtual void measure(extents_t&) { ... };  
    virtual void place(const place_data_t&) { ... };  
};  
  
struct YourWidget : PlaceableBase {  
    virtual void measure(extents_t&) { ... };  
    virtual void place(const place_data_t&) { ... };  
    ...  
};  
  
layout_engine le;  
MyWidget* m = new MyWidget();  
YourWidget* y = new YourWidget();  
...  
le.append(m); le.append(y);  
...  
le.solve();
```

OO reference semantics

- ▶ Intrusive, widget types must inherit from common base
- ▶ Ownership unclear

Our approach

- ▶ Idioms in ConceptC++ to realize run-time polymorphism in a transparent way
 - ▶ We also provide emulations in C++ 2003
- ▶ Generic components instantiated with wrappers that provide run-time polymorphism
- ▶ Non-intrusive, no changes needed to wrapped types
- ▶ ConceptC++'s adaptation mechanism (concept maps) hide wrapping from clients
- ▶ Retain “value semantics”
- ▶ Minimize overhead (small object optimization, move semantics)

Our approach: example use

- ▶ A generic layout engine, oblivious of whether it is used polymorphically or not

```
template <Placeable P> struct layout_engine { ... }
```

- ▶ Static use

```
layout_engine<HUIViewRef> le;  
HUIViewRef w;  
le.append(w); ...; le.solve();
```

- ▶ Transparent non-intrusive polymorphic use

```
layout_engine<poly<placeable>> le2;  
HUIViewRef w; MyWidget x; YourWidget y;  
le2.append(w); le2.append(x); le2.append(y)
```

- ▶ `HUIViewRef`, `MyWidget`, `YourWidget` can come from arbitrary libraries, no need for a common base class or conformance to certain function signatures

Non-intrusive adaptation with concept maps in ConceptC++

- ▶ The language we use is C++ extended with “concepts”
- ▶ Concepts are on their way to the next revision of standard C++
- ▶ Key features:
 - ▶ **requires** clause — specify constraints on type parameters
 - ▶ **concept** — a collection of requirements on a type or types
 - ▶ **concept_map** — a non-intrusive adaptation mechanism that establishes “type models a concept”
- ▶ Provides constrained templates for C++ — concept maps and overloading on constraints \Rightarrow expressive adaptation mechanism

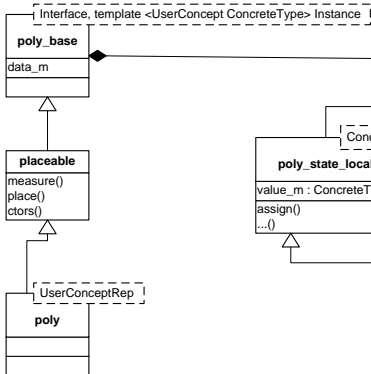
Concepts and concept maps

```
concept Placeable <typename T> : Copyable<T> {  
    void measure(T& t, extents_t& result);  
    void place(T& t, const place_data_t& place_data);  
}  
  
template <Placeable P> struct layout_engine { ... }  
  
concept_map Placeable<MyWidget> {  
    void measure(MyWidget& t, extents_t& result) { ... };  
    void place(MyWidget& t, const place_data_t& place_data) { ... };  
}  
  
concept_map Placeable<poly<placeable>> {  
    void measure(poly<placeable>& t, extents_t& result) {  
        t.measure(result);  
    }  
    void place(poly<placeable>& t, const place_data_t& place_data) {  
        t.place(place_data);  
    }  
}  
  
layout_engine<MyWidget>;  
layout_engine<poly<placeable>>;
```

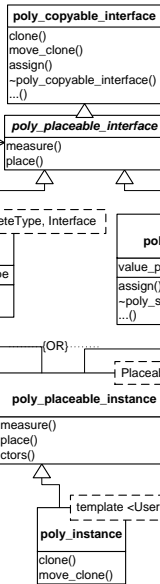
Detailed contents of the paper

- ▶ How to define wrapper types to provide non-intrusive run-time polymorphic value types in ConceptC++
 1. Implement the concept (`Placeable`) as an abstract base class
 2. Derive a generic implementation class (`PlaceableImpl<P>`) where each member function delegates to `P`, which models `Placeable`.
 3. Define a concept map `Placeable<MyWidget>` for a particular concrete type `MyWidget`
 4. Implement a handle over over the abstract base class. Make it constructible from any `Placeable` type, to be wrapped to `PlaceableImpl`
- ▶ A commonality/variability analysis that captures reusable parts to framework/library “poly.” In `poly<placeable>`
 - ▶ `poly` template provides regularity, small object optimization, move semantics
 - ▶ `placeable` provides the interface specific to the `Placeable` concept
- ▶ Concept refinement dynamically

Handle splits into 3 classes, the variability is captured in the user supplied UserConceptRep



Instance splits into 2 class templates, the variability is captured in the user supplied Instance



Interface splits into 2 classes, the variability is captured in the user supplied Interface

Related work

- ▶ Boost.Any (Henney)
- ▶ Dynamic any (Nasonov)
- ▶ Polymorphic wrappers share similarities with existential types

Conclusions

- ▶ Mixed paradigm for genuinely composable components
 - ▶ Non-intrusive
 - ▶ run-time polymorphic if so wanted: the kind of polymorphism (static, dynamic) becomes property of client of a library, not hard-coded to the interface
- ▶ Foundation of APIs of many components in Adobe Source libraries that have proven to be highly composable
 - ▶ Layout library, layout parser, property model library, property model parser
- ▶ Future directions: parameterization over concepts would be useful